A qualitative comparison study between common GPGPU frameworks.

Planning Report, Rev 0.1

Adam Söderström 930327-3750
adaso578@student.liu.se

January 2018

1 Background

The performance inclination of single-cored CPU's have during the last decades slowly started to decline. The main reason for this declination is due to three "walls":

- Instruction Level parallelism wall — not enough instruction level parallelism to keep the CPU busy
- Memory wall — gap between the CPU speed and off-chip memory
- Power wall — Increased clock rate needs more power which leads to heat problems

This has started a trend where Central Processing Unit (CPU) manufactures have started to create chips containing multiple cores that are run in parallel, see figure 1. Today modern CPU's may contain as much as 24 cores, and the number of cores available on a chip seem to be increasing. This technology is however already in use in Graphical Processing Units (GPU), which may contain hundreds of cores. This in turn have spawned a new trend among developers to not just use the GPU to render graphics to the screen, but to perform more general computations. The term used for this is General-purpose computing on graphics processing units (GPGPU).

In 2007, Nvidia released their framework called CUDA which was developed specifically for GPGPU. Since then, more frameworks and platforms have emerged, most noticeably Open Computing Language (OpenCL), OpenGL Compute Shaders and Microsoft's version of compute shaders for DirectX, DirectCompute. This thesis will focus on evaluating GPGPU frameworks when running a suitable algorithm. Performance, portability and features will be evaluated with focus on the performance. The thesis will be performed at the company MindRoad AB.

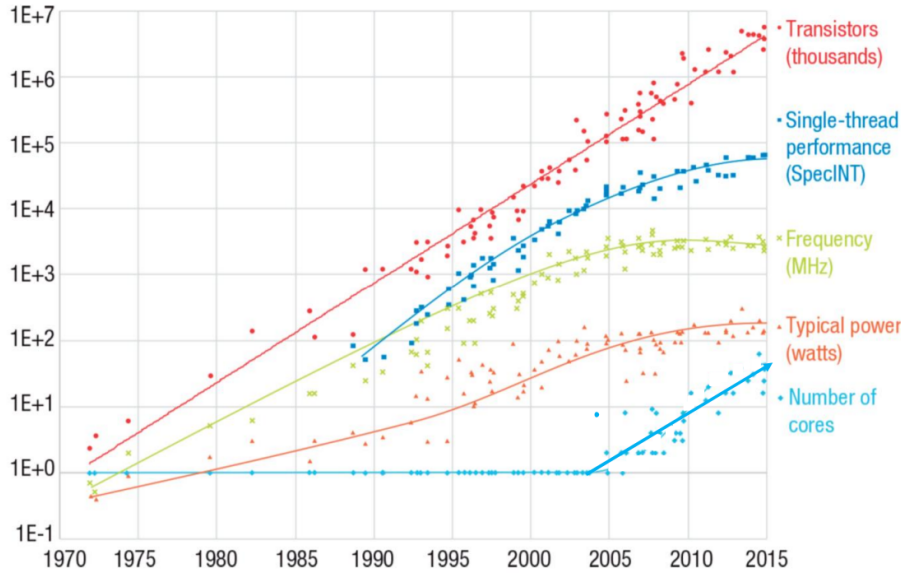

Figure 1: Statistics of development for CPU's. [6]

1.1 Algorithm

This section will discuss algorithms that are candidates for implementation and evaluation. A short description of the algorithm will be presented along with why the algorithm is suitable for a bench-marking application.

1.1.1 N-body

An N-Body simulation is an interesting implementation that can be well parallelized. A N number of bodies are simulated where each body is affected by forces from all other bodies. The traditional implementation does thus run in the time complexity $O(n^2)$ but can be further optimized by using the Barnes-Hut algorithm which uses an quad/octree the reduce the time complexity to $O(n\ log\ n)$ [1]. An N-body simulation is often used in traditional GPU bench-marking tests, and it would be interesting to investigate this further when implemented using a GPGPU approach. The bench-marking can be performed in multiple ways; in a real-time simulation and compare the frames-per-second (FPS) with the size of N. In a pre-computed simulation for a fixed amount of time-steps t_n, the bench-marking can be performed by comparing the computation time for the entire time-space.

1.1.2 Parallel Quick-Sort

The quick-sort algorithm is one of the most popular sorting algorithms. The sorting algorithm runs in the time complexity $O(n\ log\ n)$ in average, and in the

worst case $O(n^2)$. Although a very popular sequential algorithm, it is not as popular in parallel applications due to its data dependent reorganization. Some research has been done on the subject though and parallel implementations exists [8][9][7]. The bench-marking in this algorithm can be done by comparing the time consumption when sorting the same input data for all relevant frameworks.

1.1.3 Anti Aliasing using an Euclidean distance transform function

A problem in computer graphics is the inability to render sharp surface features when rendered up close, sharp edges in textures often appear jagged when rendered up close. In 2007, Chris Green of Valve Software published a method of dealing with this problem by generating a distance function for a binary image [2]. An example of C. Green's method applied to a alpha-blended texture can be seen in figure 2 [2]. S. Gustavson et. al. later released an improved version of C. Green's idea, which uses an euclidean distance transform (DT) [4][3].

(a) 64x64 texture, alpha-blended

(b) 64x64 texture, alpha tested

(c) 64x64 texture using Green's technique

Figure 2: Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) Green's distance field technique

V. Ilic et. al. later extended this into three-dimensions using a similar DT technique based on C. Green's technique [5].

The algorithm works on pixel level and is thus very well suitable for parallelization. It is also easy to use this algorithm for a bench-mark application by comparing the time consumption when running the algorithm on the different frameworks.

2 Problem formulation

- What is a suitable bench-marking algorithm?
- What factors can be compared more than the execution time?
- How does a parallel implementation compare to a sequential implementation?

3 Approach

The report will be written in parallel with the implementation and will written according to the Gantt-chart presented in figure 3. The Gantt-chart also specifies more precise time approximations and scheduling.

At the end of each week a short summary describing the work performed will be submitted to the examiner and to MindRoad AB.

- Explore previous comparisons/bench-marks between CUDA, OpenCL and DirectCompute.
- Investigate the algorithm and find parts that can be parallelized.
- Examine previous research on the subject.
- Investigate useful technologies that can be used in the implementation.
- Implement a simple "Hello World" application in all environments.
- Develop a sequential implementation used for comparison.
- Implement a CUDA application running the algorithm, perform necessary optimization's.
- Port the CUDA implementation to OpenCL and DirectCompute.
- Perform measurements between the different implementations.

4 Delimitations

This section will present some delimitations for the implementation, both in terms of the GPGPU implementation, as well as some limitations of the selected algorithm.

4.1 GPGPU

The selected algorithm will only be implemented in the discussed frameworks:

- CUDA
- OpenCL
- DirectCompute

aswell as a traditional implementation. The selected algorithm will thus not be implemented in other GPGPU frameworks such as OpenGL's compute shader, or a parallel CPU based implementation, using e.g OpenMP or similar frameworks.

4.2 N-Body

4.3 Parallel Quick Sort

4.4 Anti-Aliasing using an Euclidean distance transform function

Although a three-dimensional extention exists to C. Green's method [2], the implementation will only be performed in 2D using S. Gustavsons improved version of the technique [4].

5 Resources

The resources listed below will be used regardless of what algorithm is selected. Additinal resources such as OpenGL will be used if a visualization based algorithm is chosen.

- CUDA - http://docs.nvidia.com/cuda/
- OpenCL - https://www.khronos.org/opencl/
- DirectCompute - https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx
- Course material from the course TDDD56 - Multicore and GPU Programming

References

[1] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446, 1986.

[2] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. ACM, 2007.

[3] Stefan Gustavson. 2d shape rendering by distance fields. 2012.

[4] Stefan Gustavson and Robin Strand. Anti-aliased euclidean distance transform. *Pattern Recognition Letters*, 32(2):252–257, 2011.

[5] Vladimir Ilić, Joakim Lindblad, and Nataša Sladoje. Precise euclidean distance transforms in 3d from voxel coverage representation. *Pattern Recognition Letters*, 65:184–191, 2015.

[6] R. Stanley Williams Kirk M. Bresniker, Sharad Singhal. *Adapting to Thrive in a New Economy of Memory Abundance*. 2015.

[7] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. Cuda-quicksort: an improved gpu-based implementation of quicksort. *Concurrency and Computation: Practice and Experience*, 28(1):21–43, 2016.

[8] Peter Sanders and Thomas Hansch. Efficient massively parallel quicksort. In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 13–24. Springer, 1997.

[9] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 372–381. IEEE, 2003.

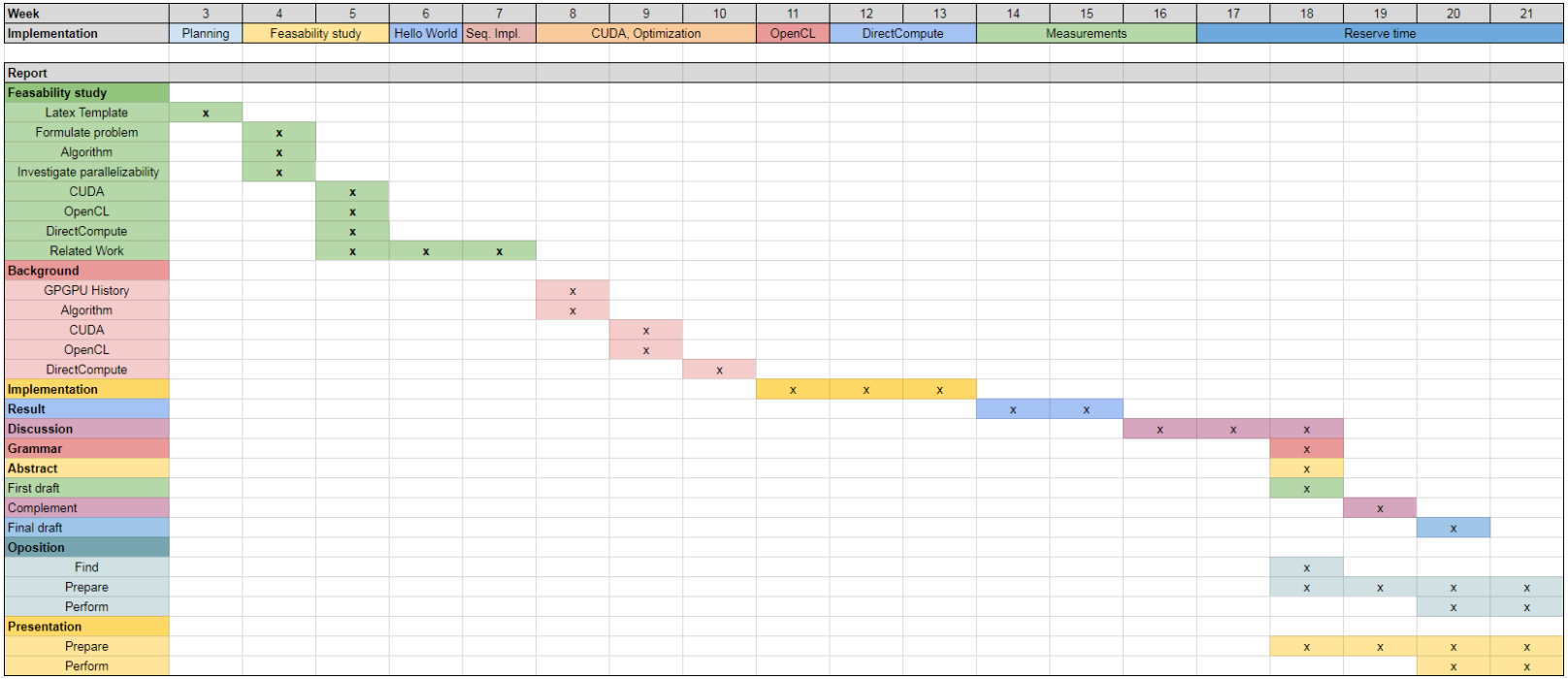

Figure 3: Detailed Gantt-chart describing the implementation and report work-flow